INTERACTIVE WEB-BASED CALCULUS TUTORIALS

An Undergraduate Research Scholars Thesis

by

MATTHEW WEIHING

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Philip B. Yasskin

May 2019

Major: CEEN

TABLE OF CONTENTS

ABSTRACT

Interactive Web-Based Calculus Tutorials

Matthew Weihing
Department of Electrical Engineering
Texas A&M University

Research Advisor: Dr. Philip B. Yasskin
Department of Mathematics
Texas A&M University

Programs exist, such as Maple, which enable developers to create tutorial applications that use a computer algebra system for complex calculus computations involved in generating problems, checking student responses and generating interactive 2D and 3D plots which are static or animated. However, only some of these applications can be embedded in a webpage.

In response, this project has developed browser-based tutorials for the MYMathApps Calculus [1] text which generate random calculus problems for students to solve, check their responses (including all intermediate steps) and give helpful feedback. Many tutorials include professional quality 2D and 3D graphics (static and animated) that are interactive in that students can plot their own functions, rotate or drag objects with a mouse, change colors (for accessibility) and add or remove objects from the plot.

Generating problems and checking answers is done using the Sage [2] computer algebra system. The graphics are made with three.js [3], a powerful WebGL-based JavaScript library. Math is displayed in LaTeX using MathJax [4]. User answers are parsed by MathLex [5] so they can be displayed in Latex and sent to the Sage server. The web page is written in HTML5, CSS and JavaScript using Node.js [6] and React [7].

ACKNOWLEDGEMENTS

I would like to thank Dr. Yasskin, my faculty sponsor for this project, who has provided guidance and input throughout this project. I would also like to acknowledge Matthew Barry, developer of the MathLex parser used in this project for his advice and guidance on this project as well. Joseph Martinsen who provided advice and input in the React portion of this project. Finally, Drew DeHaven and Akash Rao used some of the work in this project to create Tutorials for Dr. Yasskin. I appreciate all of the help that these individuals have given me.

TABLE OF FIGURES

NOMENCLATURE

MYMathApps	Meade Yasskin Math Apps
CAS	Computer Algebra System
MathJax	A JavaScript library for rendering LaTeX in a browser
MathLex	Parser to translate plaintext to LaTeX
Maple	A CAS
Sage	An open source CAS
Node.js	An open source server environment
React	A JavaScript library for developing user interfaces
WebGL	An API for rendering graphics in JavaScript
three.js	An open source JavaScript graphics library

CHAPTER I

INTRODUCTION

History

Dr. Yasskin and Dr. Douglas Meade have previously created Maplets For Calculus, a collection of over 200 Maple applications which each cover a different calculus concept. Student can generate new problems and solve those problems step by step in the Maplet. These Maplets are available to all students at Texas A&M so that they can use them as necessary. However, the Maplets are Java based and not usable on most devices.

Dr. Yasskin is also developing a textbook, MYMathApps Calculus, which is also available to all Texas A&M students. The Maplets are included in the textbook as standalone links for students to access. The Maplets themselves are not actually embedded into pages. This is the issue that inspired this project.

This Project

This project is mostly divided into two phases. First, develop browser-based Tutorials to replace the Mapets making them available on all computing devices. As stated earlier, the Maplets are not usable on all devices. Second, since there are over 200 existing Maplets, develop a process for efficiently creating new Tutorials. This process is implemented using templates.

The actual user interface is HTML, but it is created using React, a JavaScript library. JavaScript is incapable of preforming calculus operations, so the project needed a CAS. The Tutorials use the open source Sage CAS as opposed to Maple since Sage can more easily be set up to work with a web browser. All math on a page, whether hard-coded on the page, generated with the problem or typed by a user, is rendered in LaTeX using MathJax. When a student types

a math formula, it is parsed by MathLex into LaTeX for MathJax to display or parsed using JavaScript to input functions into plots. Some Tutorials involve interactive graphics. The Tutorials use three.js, a WebGL JavaScript library, to render high quality, interactive 2D and 3D graphics. The grading server is implemented using the Express framework for Node.js, and the grades database is implemented using MongoDB [8].

With this system, the Tutorials can be used in a variety of ways for a student and an instructor. Students can use them for drill and practice. Each Tutorial can generate any number of practice problems. Instructors can use Tutorials to generate examples to show in class and can also assign them as homework using the grading server and database.

CHAPTER II

TUTORIALS FOR USERS

Student Use

These interactive web-based calculus tutorials are designed to help students learn the various calculus concepts that appear in their courses. Most similar practice problem websites only have static examples that the student can do once and then run out of examples to work on. The tutorials in this project randomly generate as many practice problems as the students wish to work on. A student clicks to generate a new problem, completes each step of the problem, and elects to generate a new problem or move on to the next page of the textbook. When not in grading mode, the student can get hints or show the answers so they can learn how to do the problem and how to enter the answers. When in grading mode, students lose credit if they show the answer.

Instructor Use

Instructors can also use the Tutorials for teaching. In lecture, if an instructor needs an example to give, he/she can simply load a Tutorial, select new problem, and go from there. Also, the graphics make great lecture demonstrations.

General Overview of the User Interface

The user interface for each tutorial is a webpage in the MYMathApps Calculus textbook. See an example layout in Figure 1.

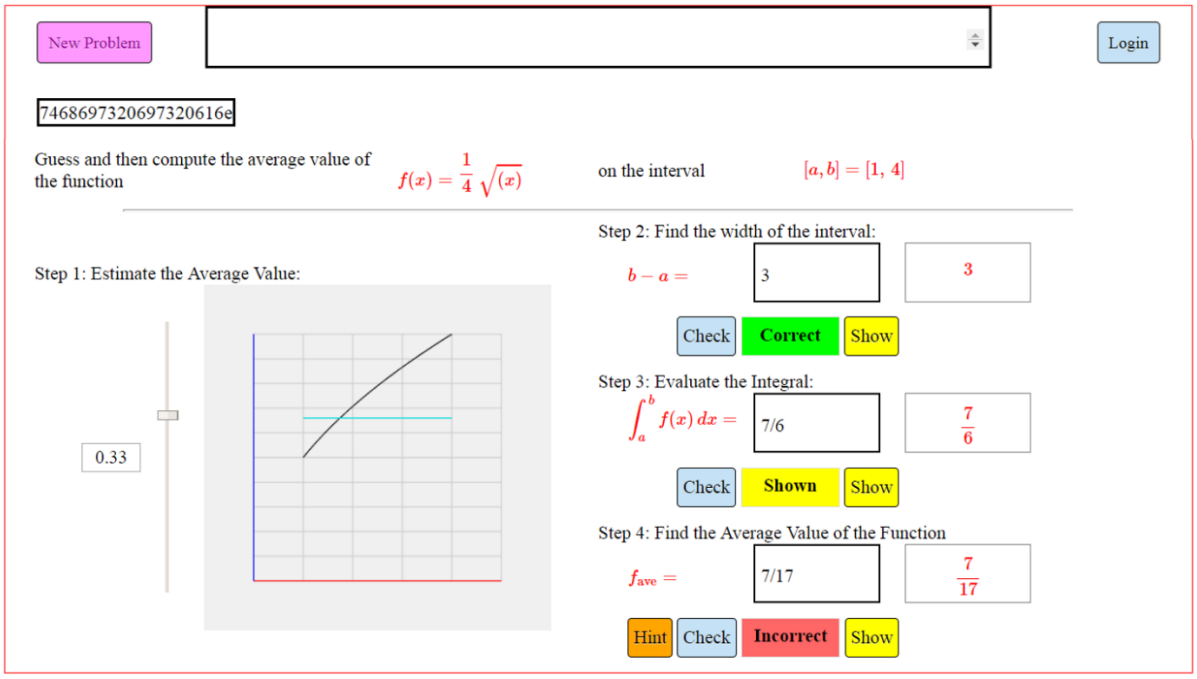


Figure 1. Example Tutorials Page

Logging In

When the student selects the optional login button, the popup window in Figure 2 will appear asking for login credentials. Currently, the Tutorials system only logs usernames, not passwords. When put into production, it will require users to register with a username and password.

Figure 2. Login Window

New Problem Generation

The student starts a tutorial by clicking on the new problem button at the top left of the page. This will call the MYMathApps server to generate a new practice problem for the student. Figure 3 shows a randomly generated problem as part of Figure 1. Alternatively, if logged in, the student can select the start grading button to generate a new graded problem. All relevant information for the student to solve the problem will then be displayed, and then the student can continue.

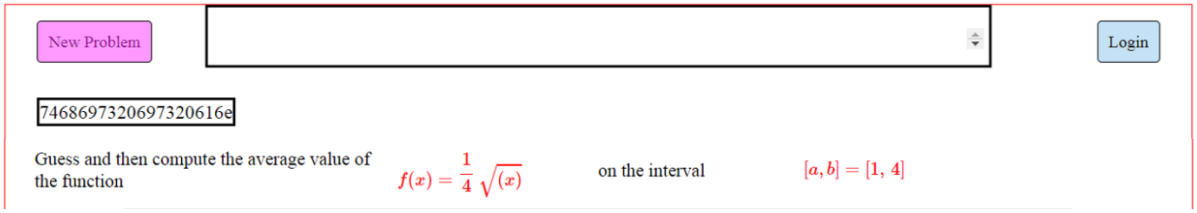


Figure 3. Example New Problem

Problem ID's

When a problem is generated, it also generates a problem ID. This ID is shown in the black-bordered box just below the new problem button (see Figure 3). At the bottom of the page is a button for problem retrieval. If the student clicks this button, the popup window in Figure 4 will appear. Pasting a problem ID in this window and clicking Retrieve will reload that problem, and the student can redo it.

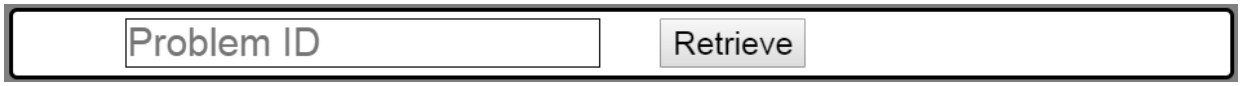


Figure 4. Problem Retrieval Window

Interactive Graphics

Some tutorials incorporate graphics to help conceptualize the topics being taught. These graphics are created when the problem is generated. These graphics are all interactive to aid the student. Figure 5 contains a graphic for students to interact with.

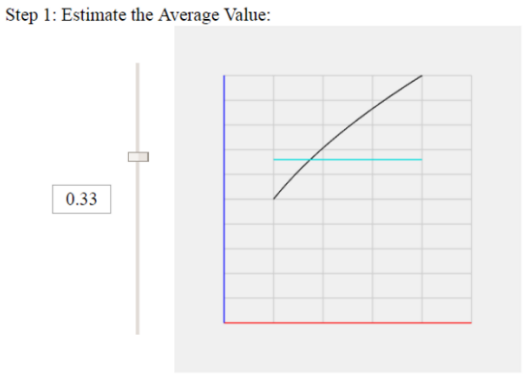


Figure 5. Example Graphic

Checking and Showing Answers

After generating a new problem, the student can begin to work on the problem. The tutorial presents a series of questions which will walk the student through the process of solving the problem step by step (see Figure 6). At each step, the Tutorial provides an instruction, for instance “Evaluate the Integral”. The black bordered box is for the student to type answers into (Figure 6). As the student types, their answer appears as “pretty print” math in the gray bordered

box to the right (Figure 6). When the student clicks “Check”, the Tutorial replies “Correct”, “Incorrect”, or “Warning” (Figure 6). The warning happens if the server could not understand the user’s answer, typically when the user’s answer contains incorrect syntax like forgetting a multiplication sign or unmatched parenthesis. The student can check an answer infinitely many times. If the student isn’t able to get the right answer, he/she can get a hint or elect to have the answer shown (Figure 6). This displays the correct answer in the input box and renders it in LaTeX.

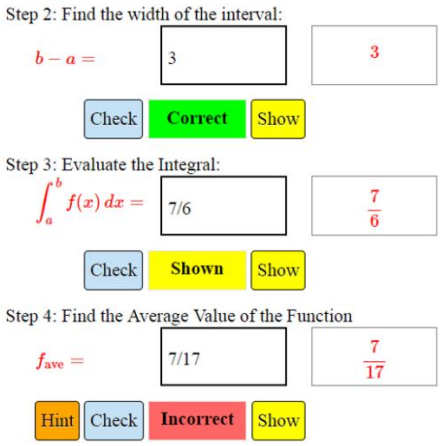


Figure 6. Example Steps for a Problem

Completed Tutorials

For this thesis, we completed four Tutorials which are in Dr. Yasskin’s textbook. They are:

- Properties of Curves
- Average Value of a Function of 1 Variable
- Volume of Revolution
- Visualizing Traces, Partial Derivatives, Tangent Lines and Tangent Planes

Each of these is discussed below.

Properties of Curves

The first Tutorial is on Properties of Curves as shown in Figure 7.

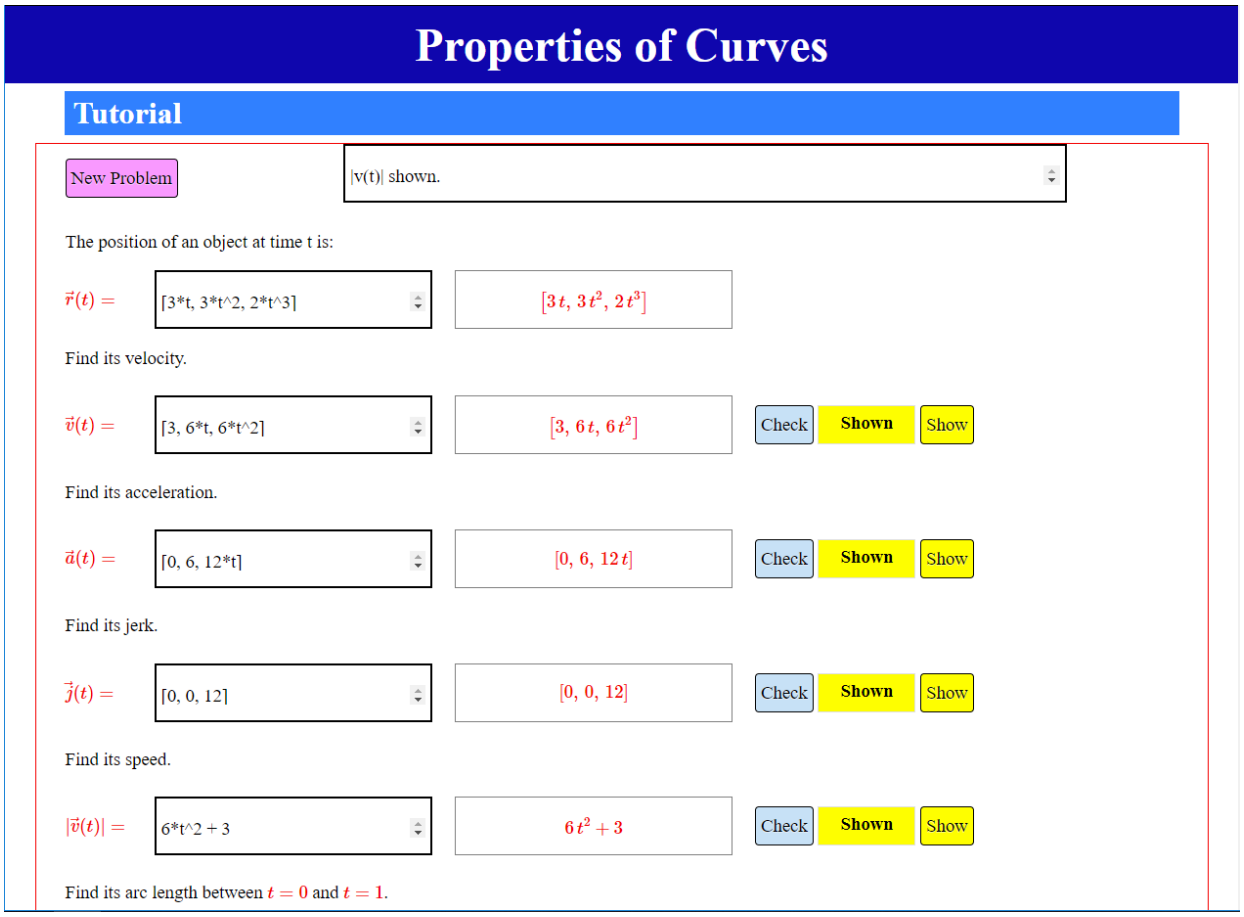


Figure 7. Properties of Curves Tutorial (Top Half)

This Tutorial generates a random position curve with respect to time and asks the students to compute the various quantities about that curve listed in the chapter. They are velocity, acceleration, jerk, speed, arc length, unit tangent vector, binormal, normal, curvature, torsion, tangential acceleration, and normal acceleration. It's not necessary but beneficial that the

student go in the order shown on the page because later steps can be performed more easily using previously solved steps.

The curves are generated so that the math simplifies easily. Most importantly, the speed is found by taking the square root of the square of velocity. The initial curve is chosen so that the square root can be easily simplified. In Figure 7, it comes out to $|v(t)| = 6t^2 + 3$. This makes the rest of the Tutorial easier for the student to complete.

Average Value of a Function of 1 Variable

The second Tutorial is Average Value of a Function of 1 Variable as shown in Figure 8. This Tutorial introduces an interactive 2D graph. Here, the student is given a function of one variable and asked to find the average value of the function. The student starts by guessing the average value by examining the graph of the curve on the left. The student either drags the slider or types in the box. Then, the student computes the length of the interval in Step 2, sets up the integral in Step 3, and evaluates it in Step 4. After completing Step 4, the actual average value is shown in the plot for the student to compare to his/her guess.

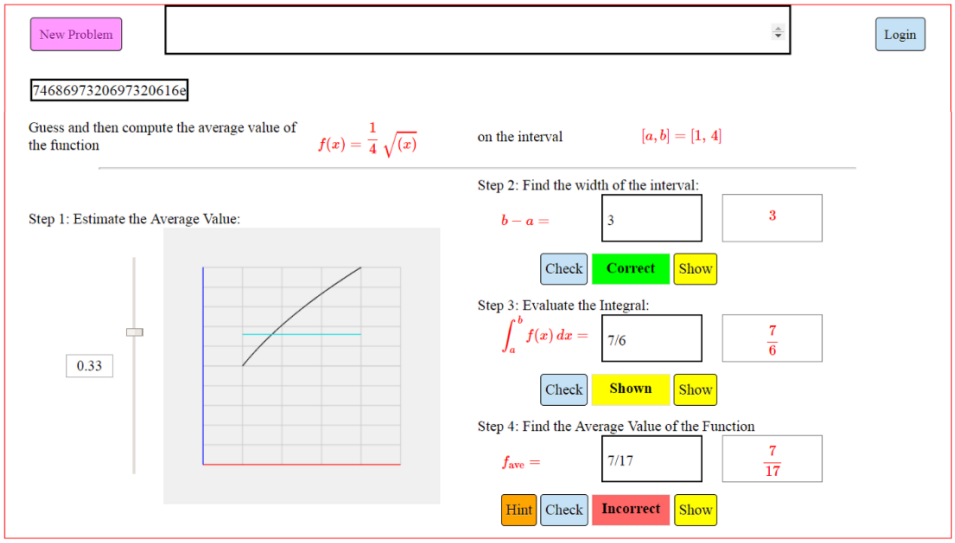


Figure 8. Average Value of a Function of 1 Variable Tutorial

Volume of Revolution

The third Tutorial is Volume of Revolution shown in Figure 9. This Tutorial introduces an animated 3D graphic which can be rotated with a mouse and also introduces hints.

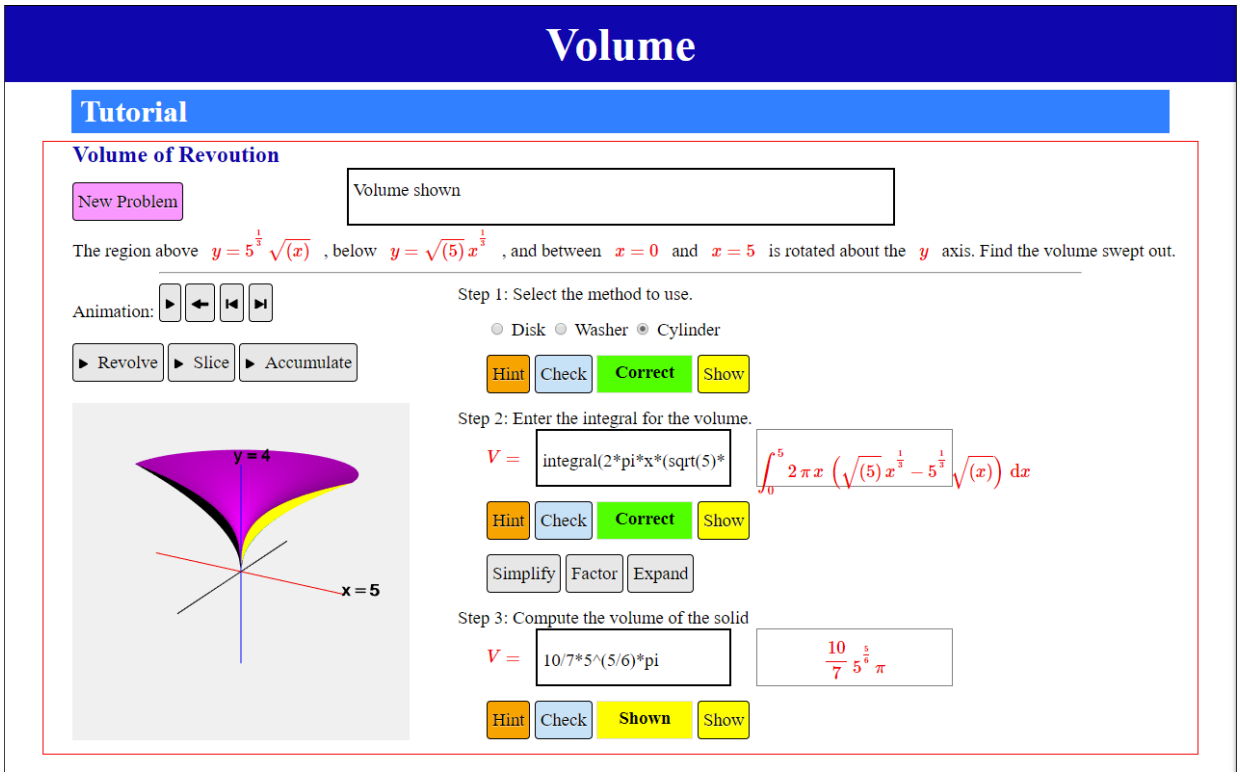


Figure 9. Volume of Revolution

This Tutorial asks the student to find the volume of the 3D figure created when a 2D region is rotated about an axis. The student can view the 3D animations by selecting Revolve, Slice, or Accumulate. Revolve (shown in Figure 9) revolves the region around a given axis. It helps the student visualize the overall figure that he/she is finding the volume of. When performing the integral to solve this problem, the student is essentially summing up the volume of an infinite number of slices. Slice animates one such slice. Finally, Accumulate animates an accumulation of those slices. Viewing the Slice or Accumulate animations helps the student

determine which method to use, the question in Step 1. Steps 2 and 3 ask the student to set up and compute the required integral. Here, the student can get a hint at each step.

Visualizing Traces, Partial Derivatives, Tangent Lines and Tangent Planes

The fourth Tutorial is Visualizing Traces, Partial Derivatives, Tangent Lines and Tangent Planes. This Tutorial is the most interactive of all the Tutorials and its features appear in a series of pages. Its first appearance introduces students to traces in Figure 10.

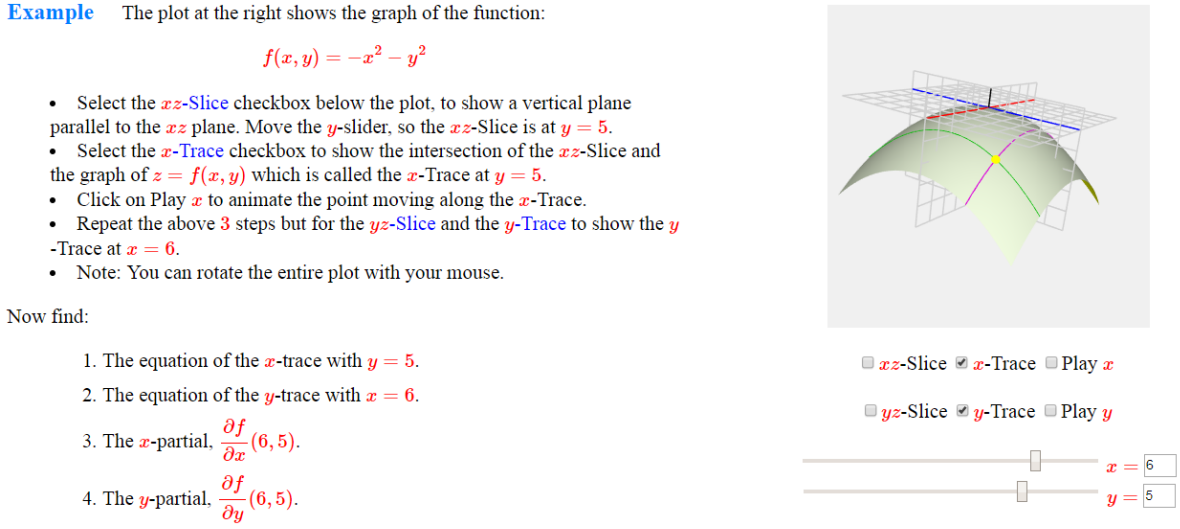


Figure 10. First Appearance of Partial Derivatives Tutorial

Here, the student can draw and move the x and y traces on a 3D surface. Students are asked to complete the tasks on the left. In the future, the bullets will change to check marks when the user completes that action. Students can use the sliders or type into the boxes to change the positions of the traces.

The Tutorial appears again adding tangent lines (in Figure 11) while also keeping the same features from Figure 10.

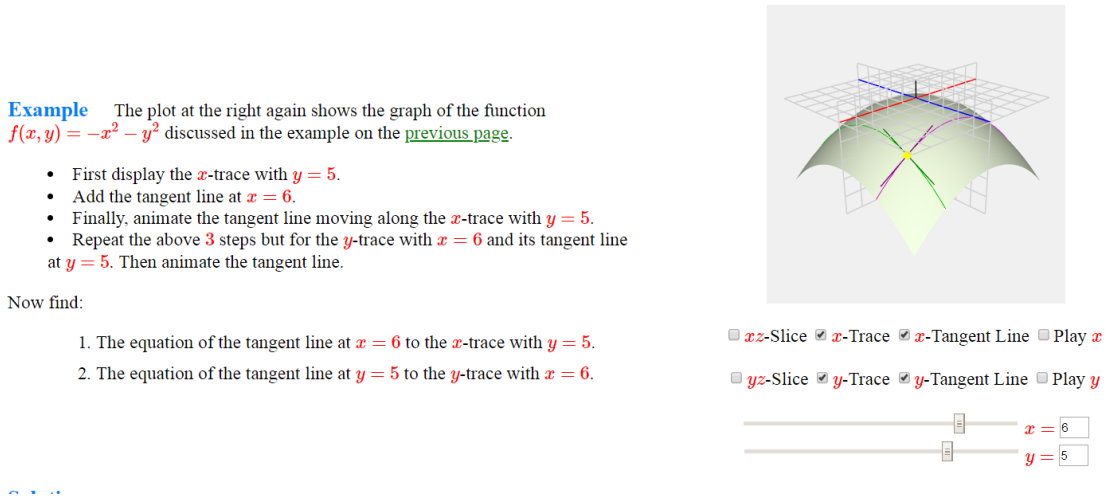


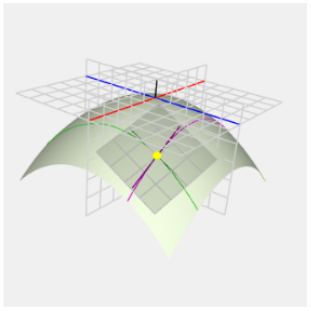
Figure 11. Introducing Tangent Lines to Partial Derivatives Tutorial

Here the student can add the tangent lines to each trace and learn that the partial derivatives are the slopes of these tangent lines.

Finally, on the next page, the Tutorial is presented a third time as in Figure 12. In this iteration, the student is able to add the tangent plane to the surface. Additionally, there is a 2D slider that the students may opt to use instead of the individual 1D sliders.

Example The plot at the right again shows the graph of the function $f(x,y) = -x^2 - y^2$ discussed in the example on the previous page.

- Display the x-trace and y-trace through $(x,y) = (6,5)$.
- Add the tangent lines to the x-trace and y-trace through $(x,y) = (6,5)$.
- Finally, add the tangent plane at $(x,y) = (6,5)$.
- You can change the point of tangency by typing values for x or y, dragging either 1D slider or dragging the 2D slider.
- You can rotate the entire plot with your mouse.



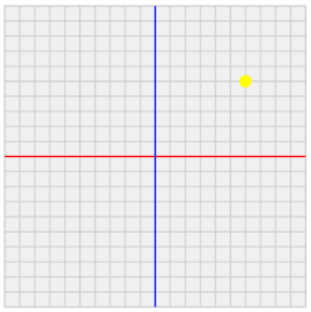
xz-Slice x-Trace x-Tangent Line Play x

yz-Slice y-Trace y-Tangent Line Play y

Tangent Plane

$x =$ 6

$y =$ 5



Now find:

1. The equation of the tangent plane at $(x,y) = (6,5)$.
2. Find the z-intercept of the tangent plane.

Solution The solution is straightforward: find the required information and plug into the formula for the equation of the tangent plane. We identify $a = 6$ and $b = 5$. We need to find $f(6,5)$, $f_x(6,5)$, and $f_y(6,5)$. It is good to make a table:

$$\begin{array}{ll} f = -x^2 - y^2 & f(6,5) = -61 \\ f_x = -2x & f_x(6,5) = -12 \\ f_y = -2y & f_y(6,5) = -10 \end{array}$$

So the equation of the tangent plane is:

$$\begin{aligned} z &= f(a,b) + f_x(a,b)(x-a) + f_y(a,b)(y-b) \\ &= f(6,5) + f_x(6,5)(x-6) + f_y(6,5)(y-5) \\ &= -61 - 12(x-6) - 10(y-5) \\ &= -12x - 10y + 61 \end{aligned}$$

We identify the z-intercept as $c = 61$.

Figure 12. Adding Tangent Planes and 2D Slider

CHAPTER III

CREATING NEW TUTORIALS

Introduction

Over 200 Tutorials already exist in Maplet form. The Maplets are not able to be used on all platforms, so it's important that they can all be converted into this new webpage format. The end goal for this project was to develop an efficient process for creating new Tutorials. The primary effort was to concentrate on a large subset of the Maplets which have similar layouts. The process is divided into two parts: the front end, i.e. the page layout, and the backend, i.e. the code executed when each button is clicked.

Creating Page Layouts

We have created two basic templates for the Tutorials, one for a single column layout, called "templateOneColumn.js", and one for a two-column layout, "templateTwoColumn.js". Since these are similar, we will only discuss the single column layout. As a model, we will show how to create the Tutorial shown in Figure 13.

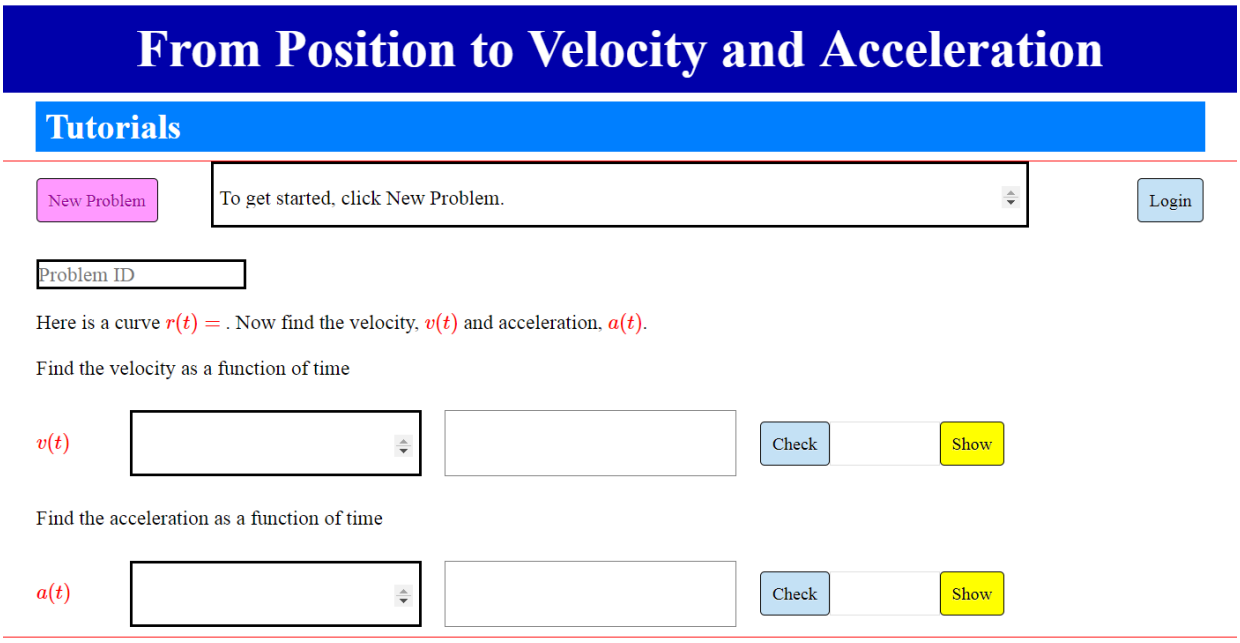


Figure 13. Tutorial Creation Example

At the top of the template, one first defines several variables as shown in Figure 14.

```
10
11  var tutorial = "PosVelAcc";
12  var tutorialTitle = "From Position to Velocity and Acceleration";
13  var problemStatement = (
14    <div>
15      Here is a curve \( r(t) = \)
16      <div id="r(t)" style={{display: "inline"}}>\( \)</div>.
17      Now find the velocity, \( v(t) \) and acceleration, \( a(t) \).
18    </div>
19  )
20  var reply="To get started, click New Problem.";
21  var steps = [
22    {
23      "step":"\\( v(t) \\)",
24      "identifier":"v(t)",
25      "instructions":"Find the velocity as a function of time","stepNumber":0
26    },
27    {
28      "step":"\\( a(t) \\)",
29      "identifier":"a(t)",
30      "instructions":"Find the acceleration as a function of time",
31      "stepNumber":1
32    }
33  ]
34
```

Figure 14. Defining Variables at the Top of the Tutorial Template

You need to enter a unique identifier for the tutorial as the variable “tutorial”, here “PosVelAcc”. Then, you enter the title as the variable “tutorialTitle”, here “From Position to Velocity and Acceleration”. Next, you enter the HTML code that defines how the problem will be displayed in the variable “problemStatement” as shown in lines 13-19 of Figure 14. There can be great variability in the style of this statement. Anywhere you want the Tutorial to insert an expression, you need to create a dynamic LaTeX display box as shown in line 16 in Figure 14. It requires a unique identifier as well as the “\(\)” as the content of the div. In this case, the Tutorial needs to be able to display the randomly generated function “r(t)”.

The “reply” is set to a default value but can be changed. Other variables, not shown, are set to default values and shouldn’t be changed. The variable “steps” contains JSON code which may be constructed as follows.

Developers can use a drag and drop interface to create the steps for a new Tutorial. This interface is shown in Figure 15.

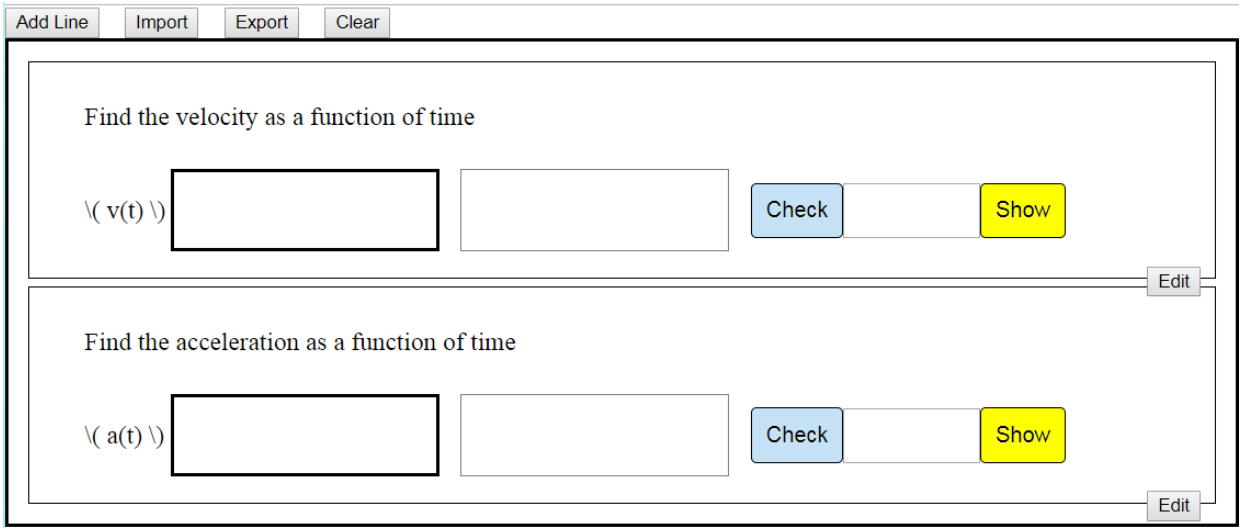


Figure 15. Drag and Drop Layout Tool

Click Add Line to add a new, blank line. The line can then be edited by pressing the Edit button. This opens up the modal in Figure 16.

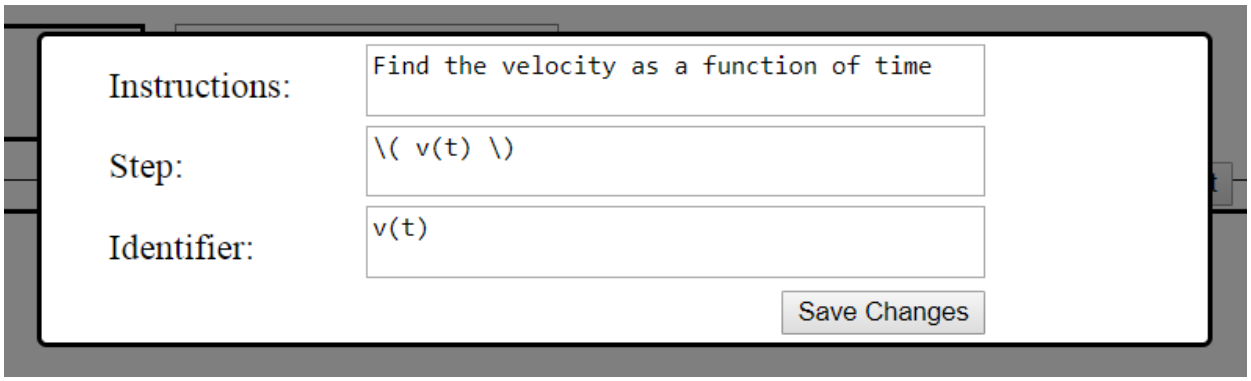
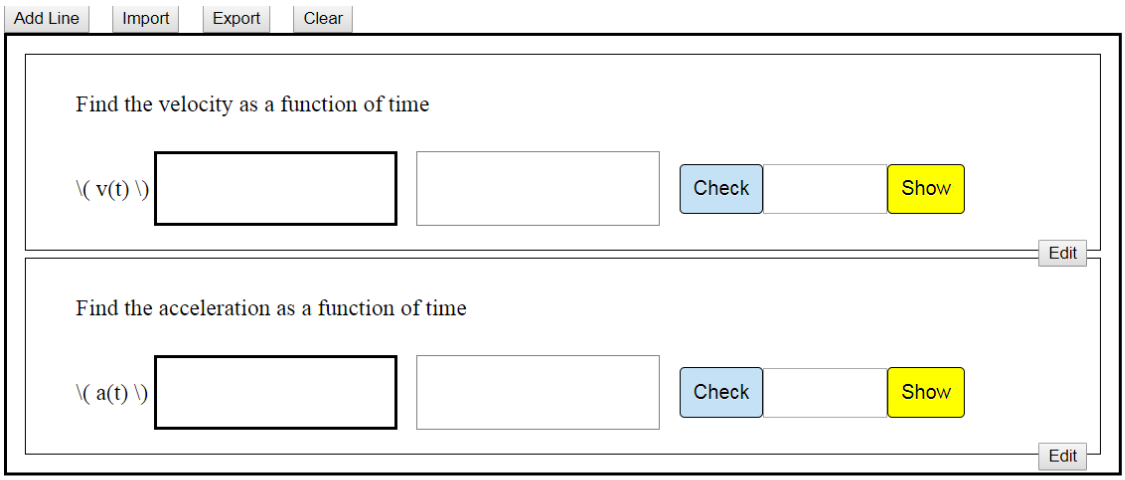


Figure 16. Edit Line Modal

The lines can also be drag and dropped to reorder them. After creating the desired layout, click Export to export the current layout into the JSON format shown at the bottom of Figure 17. The JSON string must be inserted into the template at lines 21-33 of Figure 14. The JSON can be formatted to be easier to read, but this is not necessary.



[{"step":"\\(v(t) \\)","identifier":"v(t)","instructions":"Find the velocity as a function of time","stepNumber":0},{"step":"\\(a(t) \\)","identifier":"a(t)","instructions":"Find the acceleration as a function of time","stepNumber":1}]

Figure 17. Exported Layout

Additionally, if the developer instead has the layout in JSON format, he/she can use the Import button. This open up a different modal, shown in Figure 18, which asks for a JSON object in the same format as the exported text.

Figure 18. Import Layout Modal

After placing the JSON text in the box, click Import, and the layout will now be loaded into the tool and look the same as Figure 15. This would be useful if you want to make a copy of a line and then modify it.

With the exported JSON text, the developer can now go and use the template to create a full Tutorial. At the top of the template, create a string identifier for the Tutorial for variable tutorial. Paste the JSON text and assign it to the “steps” variable. The JSON can be formatted to be easier to read, but this is not necessary.

Just below the variable section is a MathJax function which is used to set up dynamic LaTeX display boxes on the page as shown in Figure 19. Below the comment, the developer

creates new dynamic LaTeX display boxes. All that is necessary here is to inter the ID of the div, "r(t)" in this example, into line 41.

```
41 MathJax.Hub.Queue(function () {
42
43   /* Define Additional MathJax Boxes Here
44      latexBox = MathJax.Hub.getAllJax( _id )[0]
45   */
46
47   question.latexBox = MathJax.Hub.getAllJax("r(t)")[0];
48
49   for (var i = 0; i < steps.length; ++i) {
50     steps[i].latexBox = MathJax.Hub.getAllJax(i.toString())[0];
51   }
52 });
```

Figure 19. Creating Dynamic LaTeX Boxes

Further down the template, the developer needs to edit the Tutorial class. In the constructor, the template asks to set the initial values of any variables used in the problem. In Figure 20, the initial value of position is set to a blank string.

```
184  class Tutorial extends Component {
185
186    constructor(props) {
187      super(props);
188      this.state = {
189        position: "",
190        reply: reply,
191        grading: false,
192        tempUser: "",
193        user: "",
194        problemID: "",
195        userProblemID: ""
196      }
197
```

Figure 20. Setting Initial Values in Tutorial Constructor

Continue to the “displayProblem” function. This function is called in response to receiving a new problem from the server side. When setting the state, assigns values to the variables created in the constructor. In Figure 21, the position variable is set. Now, the developer must show the newly generated problem. To do this, use the “displayLatex” function to render that position function into the question latexBox.

```
210   displayProblem(response, grading) {
211     var displayFields = JSON.parse(response.data.displayFields);
212     problemID = response.data.problemID;
213
214     reply = "";
215     this.setState({
216       position: displayFields["r(t)"],
217       reply: reply,
218       grading: grading,
219       problemID: problemID
220     });
221     clearSteps();
222     displayLatex(displayFields["r(t)"], question.latexBox);
223   }
```

Figure 21. Defining Display Problem Function

Finally, define the name of the React component that this page will become in the export statement at the bottom. In Figure 22, the name of the component is PosVelAcc, the same as the Tutorial id set at the very top of the template. The two are not required to be the same.

```
356 export default class PosVelAcc extends Component {
357   render() {
358     return (
359       <div>
360         <Tutorial />
361       </div>
362     );
363   }
364 }
```

Figure 22. Setting the Name of the Tutorial's React Component

This is all the developer has to do to create a new page layout, and after running the file in React, the page will look just like in Figure 23.

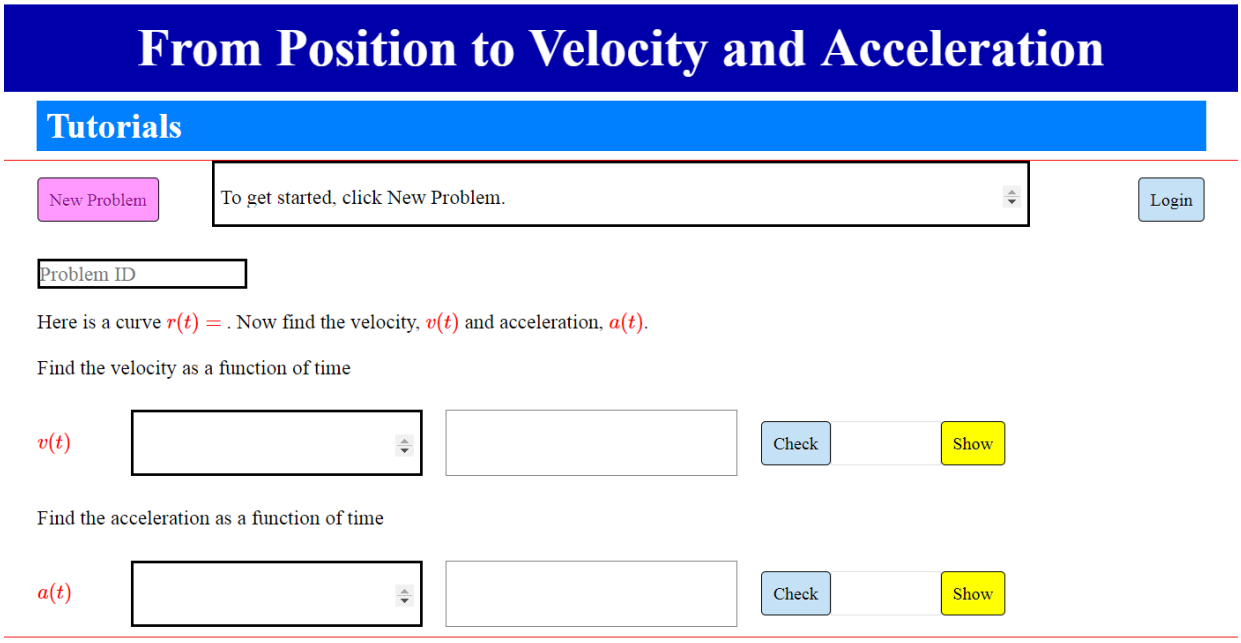


Figure 23. Completed Layout

Setting Up on Server Side

For the Tutorial to be functional, the back-end server must be set up correctly as well. On the front end, New Problem, Check, and Show are all each defined using HTTP requests to the backend, so the backend has to know what to do for this Tutorial when it receives these requests.

For New Problem generation, the developer must write how to generate new problems. This generation is written in Sage, a python-based computer algebra system. The developer can install Sage on a Linux machine to write code, or use SageCell's compiler online. The Sage code for this example is shown in Figure 24.

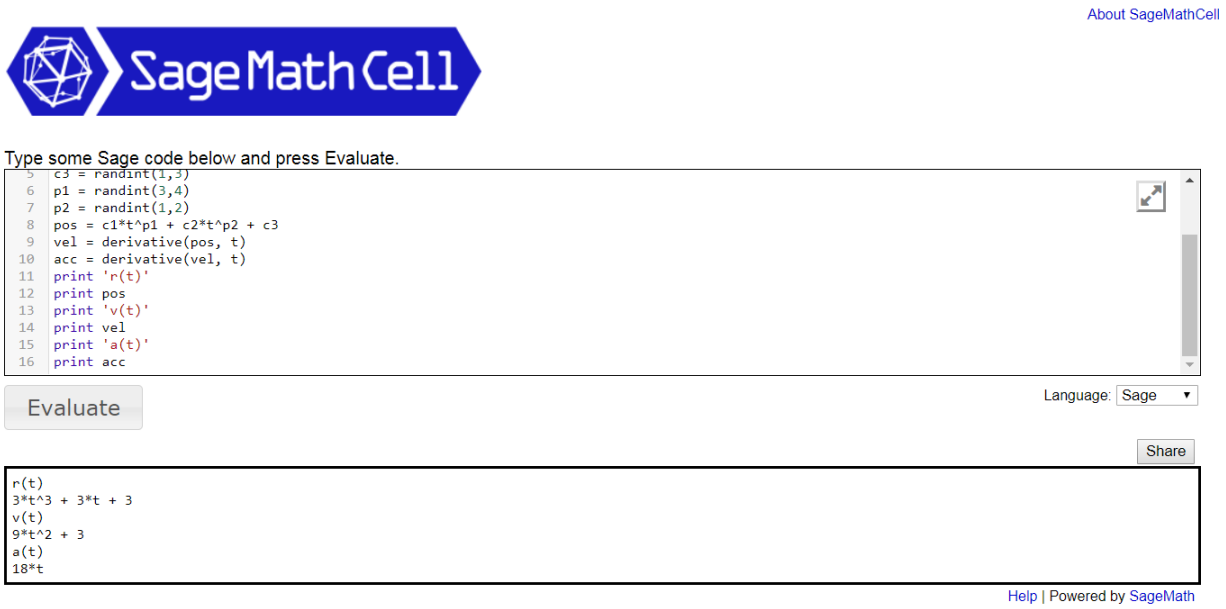


Figure 24. Developing Sage Code Using SageCell Compiler

The print statements are the methods for getting output from the Sage server. First, print the identifier to be easily pulled in JSON format. These are the same identifiers as the ones set in the front-end UI. When creating the layout, the steps were given identifiers “v(t)” and “a(t)”. It is required for these to be consistent between the Sage code and layout.

This project has a file on the server, NewProblems.js. This is where the server looks for Tutorial specific information. The file contains a JSON object containing other objects specific for each Tutorial. These objects have the format shown in Figure 25.

```
176    "PosVelAcc": {
177      title: "",
178      sageCode:"",
179      displayFields: [],
180      answerFields: []
181    }
182  }
```

Figure 25. Layout for JSON Object in NewProblems.js

The name of the object, “PosVelAcc” must be the same as the value of tutorial defined at the top of the Tutorial React page. Title is the same as the text placed in the h1 tag at the top of the HTML. For “sageCode”, place the text straight from the compiler. The text must have defined line breaks to emulate the compiler. This can be seen in Figure 26. The concatenations are not required, but separating each line makes it easier to read. The fields that the developer wishes to display go into “displayFields”, and the fields which contain answers go into “answerFields”.

```
176     "PosVelAcc": {
177       title: "From Position to Velocity and Acceleration",
178       sageCode:
179         "from random import randint\n" +
180         "t = var('t')\n" +
181         "c1 = randint(1,3)\n" +
182         "c2 = randint(1,3)\n" +
183         "c3 = randint(1,3)\n" +
184         "p1 = randint(3,4)\n" +
185         "p2 = randint(1,2)\n" +
186         "pos = c1*t^p1 + c2*t^p2 + c3\n" +
187         "vel = derivative(pos, t)\n" +
188         "acc = derivative(vel, t)\n" +
189         "print 'r(t)'\n" +
190         "print pos\n" +
191         "print 'v(t)'\n" +
192         "print vel\n" +
193         "print 'a(t)'\n" +
194         "print acc",
195       displayFields: ["r(t)"],
196       answerFields: ["v(t)", "a(t)"]
197     }
198   }
199
```

Figure 26. Completed Object for PosVelAcc Tutorial on the Server

Now the server side is complete, and the Tutorial is fully functional. A working version of it can be seen in Figure 27.

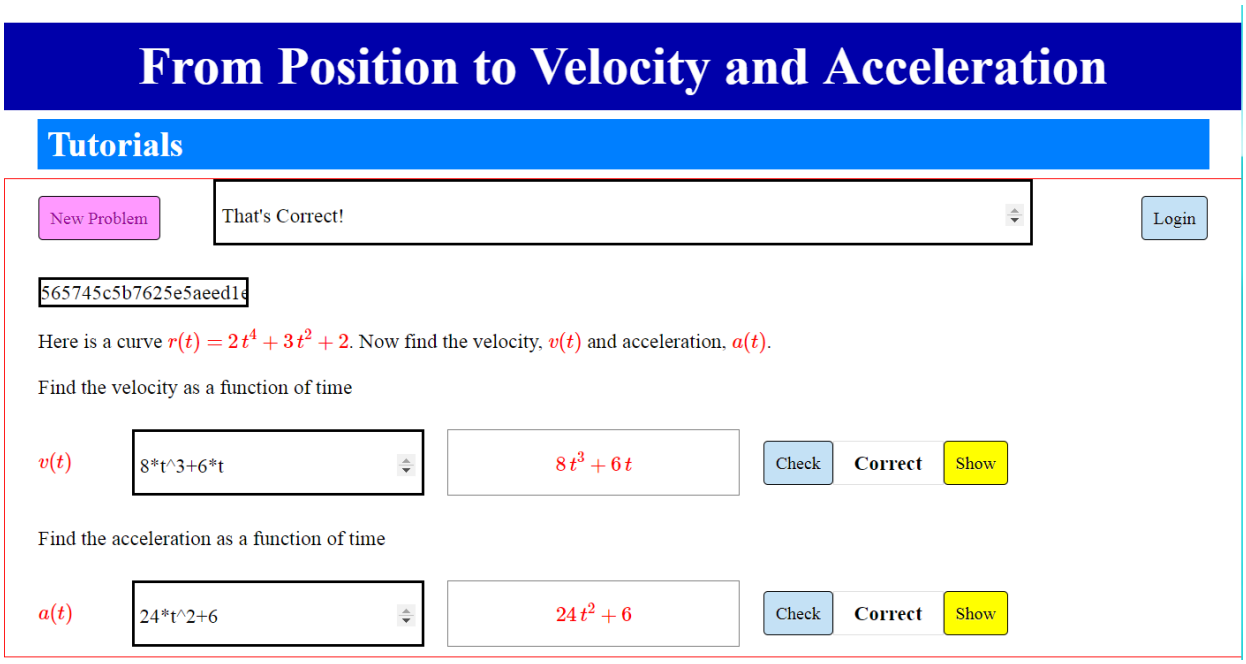


Figure 27. Functional Version of PosVelAcc

Generating Graphics

Some Tutorials contain interactive graphics which are rendered using three.js, an open source WebGL based JavaScript library. To generate graphics, the developer must first define a canvas, renderer, scene, and camera. A plot component is defined in the library. When it is placed in a page, it creates the canvas, renderer, scene, and camera, so the developer doesn't have to worry about it.

To generate figures to render, use this project's graphics library. It contains functions for generating 2D and 3D plots. After generating a figure, add it to the scene as shown in Figure 28.

```
        functionline = PlotPoints2D({
          func: functrans,
          xmin: parseInt(a),
          xmax: parseInt(b),
          ymin: 0,
          ymax: parseInt(fmax),
          step: 0.01
        })
        scene.add(functionline);
```

Figure 28. Creating and Adding a Plot

Once everything has been defined and figures are added to the scene to render, the next step is to define the render function. It calls render and then repeats recursively. This is done as shown in Figure 29.

```
function render() {
  renderer.render(scene, camera);
  requestAnimationFrame(render);
}
```

Figure 29. Defining the render Function

CHAPTER IV

DESIGNING THE TUTORIALS SYSTEM

Introduction

The entire Tutorials system in this project was designed with the idea in mind that it could easily be used to create new Tutorials. The front-end page layouts are all written in React, so that common layout can be templated and used to make more Tutorials. The grading server is written in Node.js with a JSON file containing all of the information for each Tutorial to generate New Problems. The database is implemented with MongoDB and easily expandable for new students, instructors, and Tutorials.

Software Used

The Tutorials are written as a combination of languages: HTML, JavaScript, and Sage. Page layouts are generated using React, which combines HTML and JavaScript. All math is performed by Sage, which is based on Python, and non-plaintext math is displayed on page in LaTeX. LaTeX from random problem generation and user typing is parsed using MathLex. Graphics are rendered using three.js, a WebGL JavaScript library. It's worth noting that three.js is very powerful and during testing, we ran multiple Partial Derivative Tutorial figures in the same page without any performance issues.

Designing the Front-End

Looking at the Tutorial React templates, the top of each template is identical. Import all of the packages used and declare variables. The top of the template can be seen in Figure 310 The variable "tutorial" is a unique string identifier for the Tutorial. The "tutorialTitle" is a string that will go at the top of the page. The "problemStatement" is where the developer writes the

HTML that will display the problem. Finally, “steps” is a JSON object that describes how to generate the page layout.

```
import React, { Component } from 'react';
import Popup from "reactjs-popup";
import axios from "axios";
import "./main.css";

//Basic JavaScript

declare var MathJax;
declare var MathLex;


var tutorial = /* Set Identifier for Tutorial */;
var tutorialTitle = /* Title at top of page in <h1>*/;
var problemStatement = (
  <div>
    /*
      This is where you define how to display the problem
      For instance,
      Heres a curve \( r = \) <div id="r"> \( \)</div>
    */
  </div>
)
var reply="To get started, click New Problem."
var steps = /* Paste Steps Text Here */

var tutorialObj;
var question = {};
var problemID = "";
var user = "";
var grading = false;

```

Figure 30. Top of React Template

The templates contain 3 global JavaScript functions. In order to dynamically update LaTeX, the developer must define which divs will be used for rendering LaTeX. Then, when

actually updating LaTeX on the page, call the “displayLatex” function. In it, pass the math to be rendered, and the ID of the div where the LaTeX should be rendered. That math is then parsed into LaTeX using MathLex and rendered in the LaTeX box with MathJax. Both LaTeX functions can be seen in Figure 31. Lastly, there is a function to clear all of the boxes on the page during new problem generations. It sets all of the textboxes to a blank string, and renders nothing in all of the LaTeX boxes.

```
MathJax.Hub.Queue(function () {
  /* Define Additional MathJax Boxes Here
     latexBox = MathJax.Hub.getAllJax( _id )[0]
  */
  for (var i = 0; i < steps.length; ++i) {
    steps[i].latexBox = MathJax.Hub.getAllJax(i.toString())[0];
  }
});
MathJax.Hub.processSectionDelay = 0;

function displayLatex(math, latexBox) {
  try {
    var tree = MathLex.parse(math.toString());
    var latex = MathLex.render(tree, 'latex');

    MathJax.Hub.Queue(['Text', latexBox, '\\displaystyle ' + latex]);
  }
  catch(e) {}
}

function clearSteps() {
  for (var i = 0; i < steps.length; ++i) {
    if (steps[i].line) {
      steps[i].line.setState({
        userAnswer: "",
        MR: ""
      });
    }
    if (steps[i].latexBox) {
      displayLatex("", steps[i].latexBox);
    }

    /* Clear other latex boxes created if needed */
  }
}
```

Figure 31. The Three Global Functions

Next, the templates start defining React components. The functions are the same for all line components. The only difference is the HTML. The constructors are defined as in Figure 32.

Initialize the variables to blank and bind the functions so that they can access the “this” object of the React component.

```
62  class Line extends Component {
63    constructor(props) {
64      super(props);
65      this.state = {
66        stepNumber: props.stepNumber,
67        identifier: props.identifier,
68        userAnswer: "",
69        latex: "",
70        MR: ""
71      }
72      steps[props.stepNumber].line = this;
73      this.check = this.check.bind(this);
74      this.show = this.show.bind(this);
75      this.keyupdate = this.keyupdate.bind(this);
76    }
```

Figure 32. Constructor for Line Components

The Check functions send an HTTP request to the grading server containing the information in Figure 33. Upon receiving a response, the browser displays the result. Show functions are defined in Figure 34. Again, it sends an HTTP request to the server and displays the reply

```
77    check() {
78      var thisObj = this;
79      var userAnswer = this.state.userAnswer;
80
81      axios.post('http://localhost:3001/Check', {
82        tutorial: tutorial,
83        problemID: problemID,
84        identifier: thisObj.state.identifier,
85        userAnswer: thisObj.state.userAnswer,
86        grading: grading,
87        user: user
88      })
89      .then(function(response) {
90        var miniReply = "";
91
92        var isCorrect = response.data;
93        if (isCorrect==="True\n") {
94          miniReply = "Correct";
95          reply = "That's Correct!";
96        }
97        else if (isCorrect==="False\n") {
98          miniReply = "Incorrect";
99          reply = "Your answer's not quite right. Try again.";
100       }
101       else {
102         miniReply = "Warning";
103         reply = "Oops. We didn't quite understand your answer. " +
104                 "Please check your syntax and try again.";
105       }
106
107       thisObj.setState({
108         MR: miniReply,
109       })
110       tutorialObj.setState({
111         reply: reply
112       });
113     });
114   }
```

Figure 33. Check Function for Line Components

```
116   show() {
117     var thisObj = this;
118
119     axios.post("http://localhost:3001/Show", {
120       tutorial: tutorial,
121       problemID: problemID,
122       identifier: thisObj.state.identifier,
123       grading: grading,
124       user: user
125     })
126     .then((response) => {
127       thisObj.setState({
128         userAnswer: response.data,
129         MR: "Shown"
130       });
131       reply = "Answer Shown.";
132       tutorialObj.setState({
133         reply: reply
134       })
135       displayLatex(response.data, steps[this.state.stepNumber].latexBox);
136
137       /* Again, add step specific actions if needed */
138     })
139   }
```

Figure 34. Show Function for Line Components

When a student types in an input box, it calls the “keyupdate” function shown in Figure 35. Here, it assigns the string in the input box to the userAnswer. Then, that string in rendered in LaTeX with the “displayLatex” function.

```
141     keyupdate(e) {
142       this.setState({
143         userAnswer: e.target.value,
144         MR: ""
145       });
146       displayLatex(e.target.value, steps[this.state.stepNumber].latexBox);
147     }
148
```

Figure 35. Keyupdate Function for Line Components

Lastly, define the HTML to be rendered for the component. This goes in the return statement like in Figure 36. For the basic, 1 column line, it's a single column that contains the step, a user input box, a LaTeX box for rendering what has been typed, check and show buttons, and a mini reply box.

```
149    render() {
150      return(
151        <div>
152          <div className="column">
153            <div data-col-width="8%">
154              {this.props.step}
155            </div>
156            <div data-col-width="25%" className="tutorial-input">
157              <textarea align="left" value={this.state.userAnswer}
158              onChange={this.keyupdate}></textarea>
159            </div>
160            <div data-col-width="2%"></div>
161            <div data-col-width="25%" className="tutorial-display"
162              id={this.state.stepNumber.toString()}>\[ \]</div>
163            <div data-col-width="2%"></div>
164            <div data-col-width="38%">
165              <p>
166                <button className="btn-check" onClick={this.check}>Check</button>
167                <textarea className="tutorial-feedback" rows="1"
168                  value={this.state.MR} readOnly></textarea>
169                <button className="btn-show" onClick={this.show}>Show</button>
170              </p>
171            </div>
172          </div>
173        </div>
174      )
175    }
```

Figure 36. HTML for 1 Column Line Component

The last portion of the template is the section which defines the entire Tutorial component. This is the component that actually renders the line components. Like before, the constructor initiates values and binds functions.

The first function, “displayProblem”, defines how to display a problem received from the grading server in Figure 37. It takes an object containing the fields to display and the problem ID and displays accordingly. They each have their own boxes to be displayed in.

```
207    displayProblem(response, grading) {
208      var displayFields = JSON.parse(response.data.displayFields);
209      problemID = response.data.problemID;
210
211      reply = "";
212      this.setState({
213        /* Set Values of Display Fields */
214
215        reply: reply,
216        grading: grading,
217        problemID: problemID
218      });
219      clearSteps();
220      /* Display Latex Appropriately
221         use function displayLatex
222      */
223    }
```

Figure 37. Function to Display Tutorial Problem

To actually create the problems to display, either “newProblem” or “startGrading” is called in Figure 38. Each of the functions correspond to a button. The new problem button starts an ungraded problem, and the start grading button begins a graded attempt. The “generateNewProblem” function sends a request to the server containing the tutorial ID, whether the problem is graded, and the user. It sends the response to the “displayProblem” function.

```
225    generateNewProblem(toGrade) {
226      var thisObj = this;
227      grading = toGrade;
228
229      axios.post('http://localhost:3001/New', {
230        tutorial: tutorial,
231        grading: grading,
232        user: user
233      })
234      .then(function(response) {
235        thisObj.displayProblem(response, grading);
236      });
237    }
238
239    newProblem() {
240      this.generateNewProblem(false);
241    }
242
243    startGrading() {
244      this.generateNewProblem(true);
245    }
```

Figure 38. Creating New Tutorial Problem

Instead of generating a new problem, a student may wish to complete a previously problem. A student might have had trouble with this problem and would like to ask an instructor for help. In this case, the student can retrieve a problem with the problem ID. This function, shown in Figure 39, sends the ID to the server and displays the problem in the same way a randomly generated problem is shown.

```
247     retrieveProblem() {
248       var thisObj = this;
249       console.log(this.state.userProblemID);
250
251       axios.post('http://localhost:3001/Retrieve', {
252         tutorial: tutorial,
253         problemID: this.state.userProblemID
254       })
255       .then(function(response) {
256         thisObj.displayProblem(response, false);
257       });
258     }
```

Figure 39. Retrieving Tutorial Problem

The final function is used to generate the lines for the Tutorial defined in Figure 40. It iterates through the steps object generated by the layout creation tool creating a new Line component and displaying the instructions if necessary.

```
272     generateLines() {
273       var lines = [];
274       for (var i = 0; i < steps.length; ++i) {
275         if (steps[i].instructions) {
276           lines.push(<p>{steps[i].instructions}</p>);
277         }
278         lines.push(
279           <Line
280             step={steps[i].displayField}
281             identifier={steps[i].identifier}
282           />
283         )
284       }
285       return lines;
286     }
```

Figure 40. Function to Generate Line Components

For the HTML code of the entire Tutorial, the top of the page is always defined as shown in Figure 41. The name of the Tutorial and the word Tutorials are in h1 and h2 tags respectively. From there, the new problem button and reply box are simple and always shown. However, there is a login button with associated modal for students to use to log in. Additionally, after logging in and based on the status of the problem, currently being graded or not, wither the start grading button or the submit grade button will be shown. Finally, there is a box to display the problem ID on the next line.

```
314     return (
315       <div>
316         <h1> {tutorialTitle} </h1>
317         <h2>Tutorials</h2>
318         <div className="tutorial">
319
320           <div className="column">
321             <p data-col-width="15%">
322               <button className="btn-primary" onClick={this.newProblem}>
323                 New Problem
324               </button>
325             </p>
326             <div data-col-width="70%" className="tutorial-message">
327               <textarea rows="2" value={this.state.reply} readOnly></textarea>
328             </div>
329             <div data-col-width="15%" align="right">
330               <Popup trigger={<button className="btn-check"
331                 hidden={!this.state.user == ""}>Login</button>}
332                 contentStyle={{borderRadius: "5px", border: "2px solid black"}}
333                 modal
334               >
335                 { close => (
336                   <div align="center">
337                     <div style={{float: "left"}}>     
338                       <input type="text"
339                         placeholder="Username" value={this.state.tempUser}
340                         onChange={this.handleUsernameInput}
341                         style={{border: "1px black solid", fontSize: "24px",
342                                 height: "30px"}}/>
343                     </div>
344                     <div style={{float: "left"}}>  
345                       <button onClick={() => {user = this.state.tempUser;
346                                               this.setState({user: user}); close();}}
347                         style={{fontSize:"20px", height: "30px"}}>Login</button>
348                     </div>
349                   </div>
350                 )}
351               </Popup>
352               <button className="btn-check" onClick={this.startGrading}
353                 hidden={this.state.grading || this.state.user == ""}>
354                 Start Grading</button>
355               <button className="btn-check" onClick={this.submitGrade}
356                 hidden={!this.state.grading || this.state.user == ""}>
357                 Submit Grade</button>
358             </div>
359           </div>
360           <p>
361             <input type="text" value={this.state.problemID}
362               className="problemID" readOnly
363               style={{resize: "none", border: "2px solid black"}}
364               placeholder="Problem ID"
365               onFocus={(event) => event.target.select()}/>
366           </p>
367
```

Figure 41. Header for Every Tutorial Page

Finally, the Tutorial component places the “problemStatement” as defined at the top and calls the “generateLines” function to create all of the Line components and places them under the problem display in Figure 42.

```
364                 placeholder="Problem ID"
365                 onFocus={(event) => event.target.select()}/>
366             </p>
367
368             { problemStatement }
369             { this.generateLines() }
370
371           </div>
372         </div>
373       )
374     }
```

Figure 42. Where to Define Problem Specific HTML

Designing the Grading Server

The server is written using the Express package for Node.js. The server handles server calls for problem generation, checking and showing answers, retrieving problems, and interacting with the grades database. Its purpose is to eliminate student’s abilities to cheat grades.

New Problem Generation

New problem requests contain the ID specific to the Tutorial, whether or not the problem is graded, and the user creating the new problem. The server then pulls the Sage code to generate a new problem from the JSON file. In order to evaluate Sage code, it must be sent to a Sage server. This project uses SageCell’s public server to evaluate the code. The Sage code is written so that it prints the name of the variable before printing a value as shown in Figure 43.

```
82        "print 'r(t)'\n" +
83        "print [r[0],r[1],r[2]]\n" +
84        "print 'v(t)'\n" +
85        "print [v[0],v[1],v[2]]\n" +
86        "print 'a(t)'\n" +
87        "print [a[0],a[1],a[2]]\n" +
```

Figure 43. Sage Code for Printing Values

The code in Figure 43 returns a string separated by new line characters, so the server splits that string by new line characters and places the strings in an array. After creating an array, it loops through the array and places the values into a JSON object. An example of this can be seen in Figure 44.

```
222  string = "r(t)\n[t,t^2,t^3]\nv(t)\n[1,2*t,3*t^2]"
223
224  array = ["r(t)", "[t,t^2,t^3]", "v(t)", "[1,2*t,3*t^2]"]
225
226  json = {
227    "r(t)": "[t,t^2,t^3]",
228    "v(t)": "[1,2*t,3*t^2]"
229  }
```

Figure 44. Example Translating Sage Response into JSON Object

This new problem JSON object is used to generate the problem id for that specific problem. The object converted into a string and that string is encrypted using the ABS-256-CBC algorithm in the npm crypto package.

In the server JSON file, each Tutorial specific object contains the field “displayFields”. This is an array that indicates which fields in that new problem object are okay to send back to

the client. These fields do not include answers, only the information required to display the problem. The field names and their values are stored in a new JSON object. In the case of the Properties of Curves Tutorial, only the function "r(t)" is returned to the student's browser because that is all the problem info needed.

If the student elected to start a graded problem, the server removes the current active Tutorial, if there was one, and starts a new active Tutorial in the database. All answer fields, which are specified in the JSON file, are initialized to 0, and the database stores which Tutorial it is and the problem ID.

Finally, the server sends back a response which contains the JSON object of display fields and the problem ID back to the browser.

Checking Answers

The check server call receives the Tutorial ID, problem ID, question specific ID, student's answer, whether the question is graded, and the user. Checking answers is handled by the server to prevent cheating.

The server decrypts the problem ID and parses back into the JSON object containing all of the answers for the problem. It takes the student's answer and compares it to the correct answer using the Sage server. If the problem is graded and the student gets the answer correct, it stores a 1 in the database for that part of the Tutorial unless the answer was previously shown. There is no penalty for an incorrect answer, and the student has unlimited attempts. If the attempt was not graded, the server does not interact with the database. Finally, the server sends a response to inform the student if the answer was correct.

Showing Answers

For showing answers, the request contains the Tutorial ID, problem ID, question specific ID, whether the question is graded, and the user.

Just like for checking answers, the problem ID is decrypted and parsed into a JSON object. If the question is graded, the server stores a -1 in the database for that part unless the student previously got the answer correct. In some instances, the student might get the answer correct and then try to simplify it. If the student fails to simplify it, he/she might wish to show the answer again. There is no penalty for this. The server sends a response containing the correct answer back to the browser.

Submitting a Problem

Starting a graded problem does not mean that the grade will ultimately be stored. The student still has to submit the problem. Upon submitting a problem, the server removes the active Tutorial in the database and appends it to the list of previous attempts for that specific Tutorial. The student will no longer have an active Tutorial. The server also updates the highest grade that the student has achieved for that Tutorial if necessary.

Retrieving a Problem

The student can elect to retrieve a problem with its problem ID. The server parses the problem ID and begins a new, not graded problem. The process is identical to the new problem generation process after creating the JSON object from the Sage response.

Designing the Grades Database

The database is implemented in MongoDB with the purpose of storing grades for students. It contains an object for each user, student or instructor. Each student has an object for each Tutorial. For example, Properties of Curves and Average Value of a Function of 1 Variable.

This Tutorial specific object contains a list of graded attempts and the highest grade of those attempts, shown in Figure 45. Additionally, it stores an active Tutorial with its Tutorial ID, problem ID, and question progress shown in Figure 46.

```
"_id" : ObjectId("5c76ff068e4c972260e5724a"),
"user" : "mcweihing",
"classification" : "student",
"CurveProps" : {
        "highestGrade" : 14,
        "attempts" : [
                {
                        "v(t)" : 1,
                        "a(t)" : 1,
                        "j(t)" : 1,
                        "|v(t)|" : 1,
                        "L" : 1,
                        "T" : -1,
                        "vxa" : 1,
                        "|vxa|" : 1,
                        "B" : -1,
                        "N" : 1,
                        "kappa" : 1,
                        "tau" : -1,
                        "aT" : -1,
                        "aN" : 1
                },
                {
                        "v(t)" : 1,
                        "a(t)" : 1,
                        "j(t)" : 1,
                        "|v(t)|" : 1,
                        "L" : 1,
                        "T" : 1,
                        "vxa" : 1,
                        "|vxa|" : 1,
                        "B" : 1,
                        "N" : 1,
                        "kappa" : 1,
                        "tau" : 1,
                        "aT" : 1,
                        "aN" : 1
                }
        ]
},
"AvgVal1var" : {
        "highestGrade" : 0,
        "attempts" : [ ]
},
```


Figure 45. Attempts Stored for CurveProps Tutorial

```
        "activeTutorial" : {
                "tutorial" : "",
                "problemID" : "",
                "progress" : [ ]
        }
```

Figure 46. Active Tutorial

For instructors, it only stores a list of students. The instructor can then look up the grades of the students in the instructor's list. An instructor is shown in Figure 47.

```
{
        "_id" : ObjectId("5c76ff068e4c972260e5724c"),
        "user" : "instructor1",
        "classification" : "instructor",
        "students" : [
                "mcweihing",
                "student2"
        ]
}
```

Figure 47. Instructor Stored in the Database

CHAPTER V

CONCLUSION

As it stands, the Tutorials system is a fully functional, standalone homework and practice problem system. Using React, it is possible to generate any user interface. So far, only basic page layouts have been made because they are the most common and the templates can be used to quickly create large amounts of new Tutorials. Creating new Tutorials on the server side is simple. All a developer has to do is add a new entry into the JSON file.

Broader Impact

The Tutorials can have a lasting impact on students. The 3D graphics can help students learn abstract concepts that instructors can't demonstrate on a whiteboard. Additionally, learning the concepts in the Tutorials are often prerequisites for other courses. It's important that the students have strong foundations.

While the Tutorials are currently only calculus-based, the system could easily be expanded into other STEM areas. For instance, physics and chemistry have many concepts that are just performing mathematical operations. This is an area that the Tutorials could thrive in.

Moving Forward

While the system is functional, it is not ready for production release. It was originally written for functionality, not security. For example, it would be beneficial to require server requests to come only from the MYMathApps domain. This reduces cheating. This hasn't been implemented, however, because the grading server is not on that domain yet.

Also, this project will be handed to other students working under Dr. Yasskin to continue. This project has only created a few Tutorials, and somebody else will create the rest. These Tutorials will be placed into the pages of his MYMathApps Calculus books.

REFERENCES

1. Meade Yasskin Math Apps, “MYMathApps.” https://ww.mymathapps.com.

2. Sagemath, “Sage.” http://www.sagemath.org.

3. “three.js” https://threejs.org.

4. MathJax, “MathJax.” https://www.mathjax.org.

5. MathLex, “MathLex.” mathlex.org.

6. Node.js, “Node.js.” htts://nodejs.org.

7. React, “React.” https://reactjs.org.

8. MongoDB, “MongoDB.” https://www.mongodb.com.